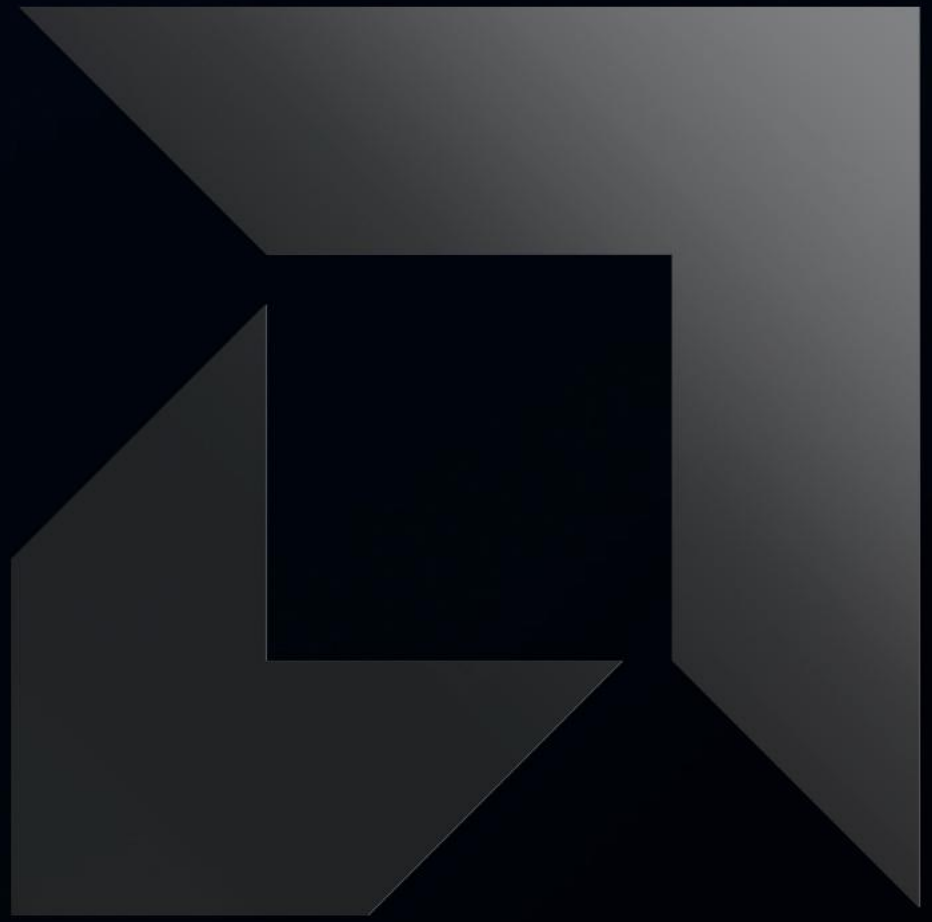


# APU Programming Model

Presenters: Bob Robey
AMD @ Tsukuba University

AMD
together we advance_

# Porting code from CPU to GPU

It is **easier than ever** to port to GPUs

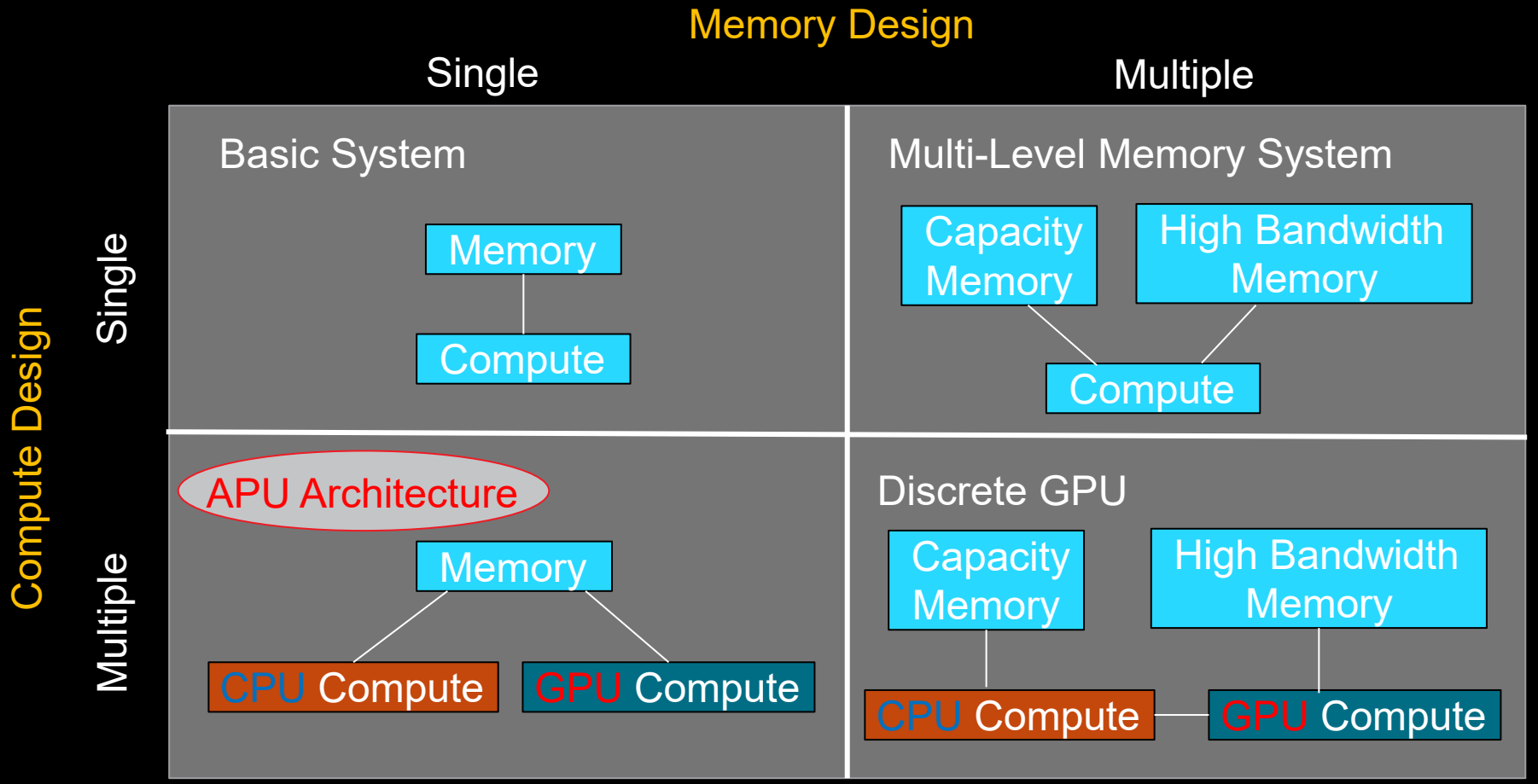
- Accelerated Processing Units (APUs) simplify porting
  - APU architecture and APU Programming model – what are these?
- Pragma-based language makes porting **quicker and more portable** to run on CPU and GPU
- **Optimized libraries for the GPU** can help
- **Interoperability** of different programming models
- New tools and **improved functionality**

# What AMD offers

- **MI300A is an APU Architecture – in the actual hardware**
  - Same memory space for CPU and GPU
  - **APU is an AMD term; others may call it an integrated GPU**
- Commitment to **APU programming model** for GPU products
  - **APU programming model applies to discrete GPUs as well**
- AMD values customer friendly policies
  - Open-source
  - No vendor lock-in
  - Portability
- Industry leading CPU performance
- Commitment to HPC customers
  - Leading FP64 (IEEE-754) performance
  - MI300A and MI300X FP64 performance increased over the MI250X products
  - No special tricks to get FP64 for general applications


# APU and discrete GPU architectures

# Taxonomy of compute architectures

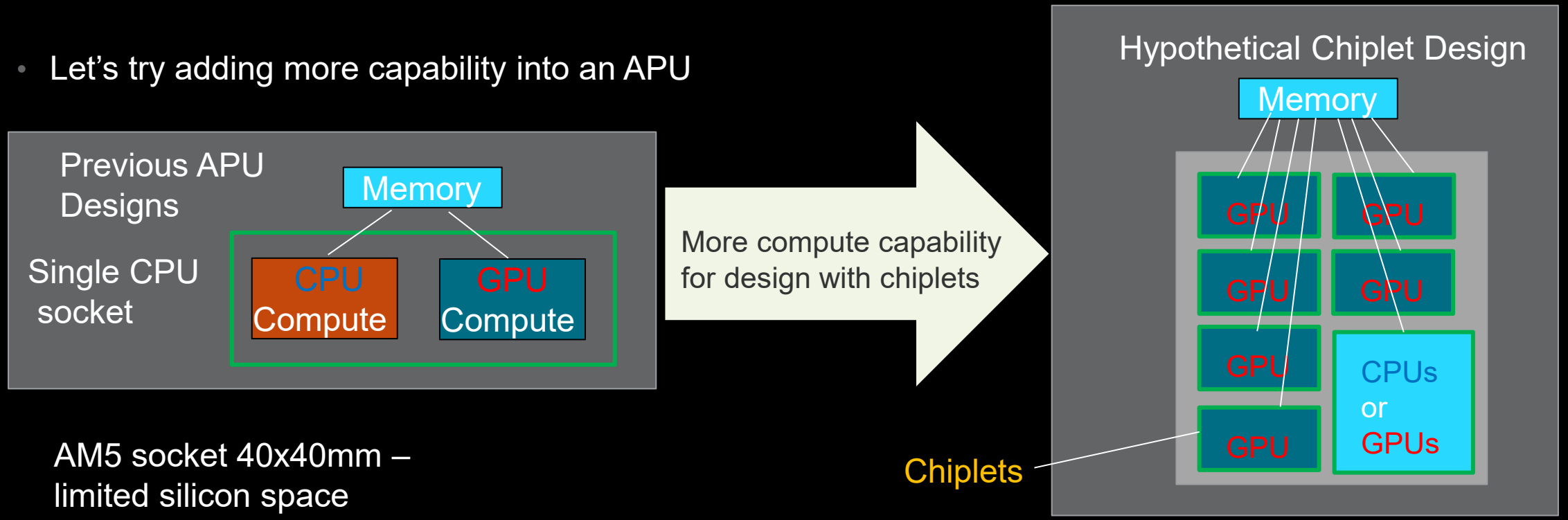



Taxonomy categorizes architecture by dominance of hardware components

- Memory Dominant – architecture revolves around a single memory space
- Compute Dominant – architecture centered around a single compute resource
- APU is primarily characterized by compute units being able to address all memory

# Breakthrough in compute capability -- Conceptual

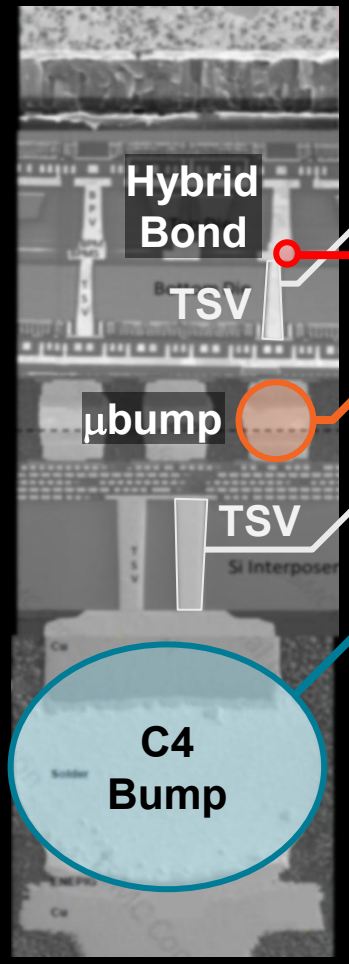
- Integrated GPUs have traditionally been limited by how much GPU compute capability can be included
  - ❖ Silicon Chip only has so much space
  - ❖ Chiplets allow us to expand that space
- Let's try adding more capability into an APU

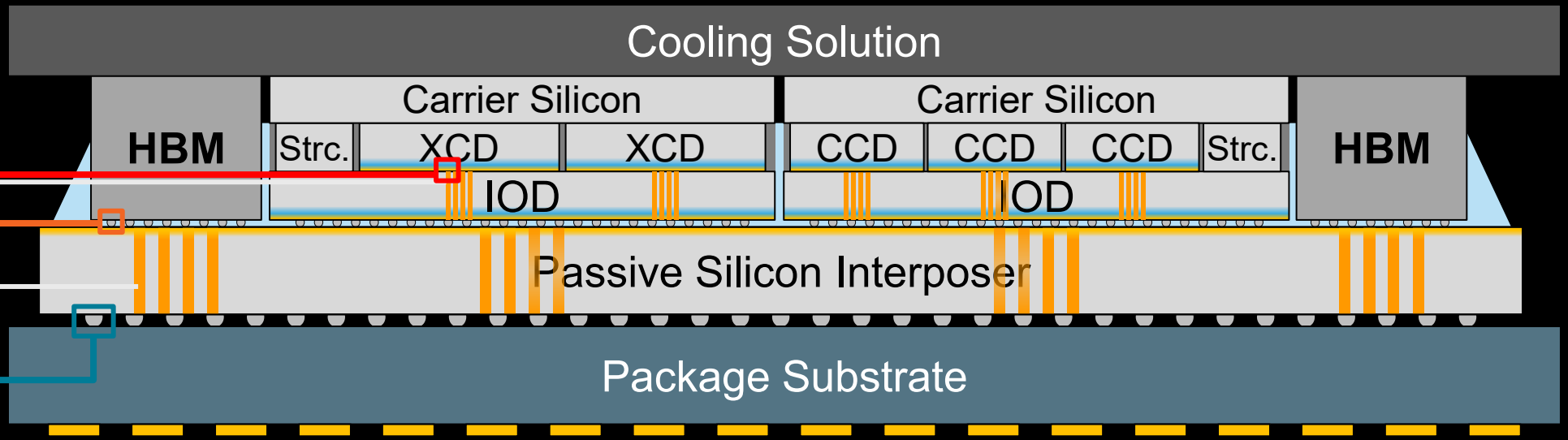



AM5 socket 40x40mm – limited silicon space

Lots more silicon space to work with

# Advanced Heterogeneous Integration Packaging



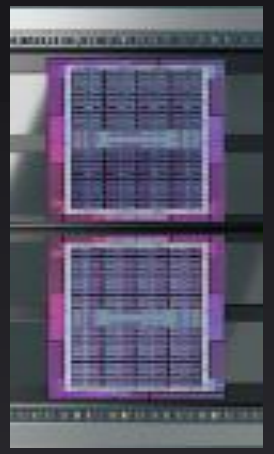




- 8 stacks of HBM
- 6 XCDs
- 3 CCDs
- 4 IODs
  - IOD – I/O Die

- 3D hybrid bonding
- 2.5D silicon interposer
  - IOD-IOD links
  - IOD-HBM links

# Bringing it to AMD Instinct™ Accelerator Products

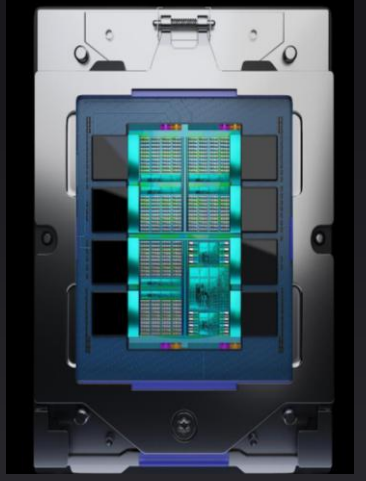
## MI200 Series



**Extreme Compute Architecture with leading memory capacity and bandwidth**

- Technology in first Exascale systems
- High compute to power ratio
- Tight integration with memory
- Infinity Fabric™ for data transfers

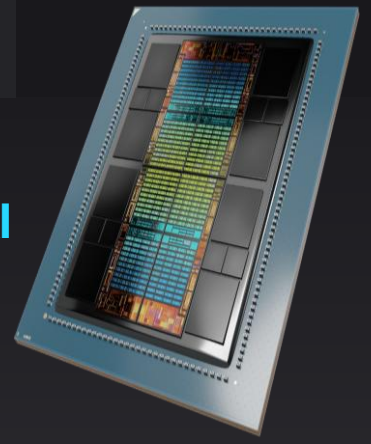
## MI300A



**First True APU Architecture for HPC and AI**

- Memory Bandwidth Workloads
- Hybrid CPU + GPU Capability
- GPUs can drive full bandwidth

## MI300X



**Leadership generative AI accelerator**

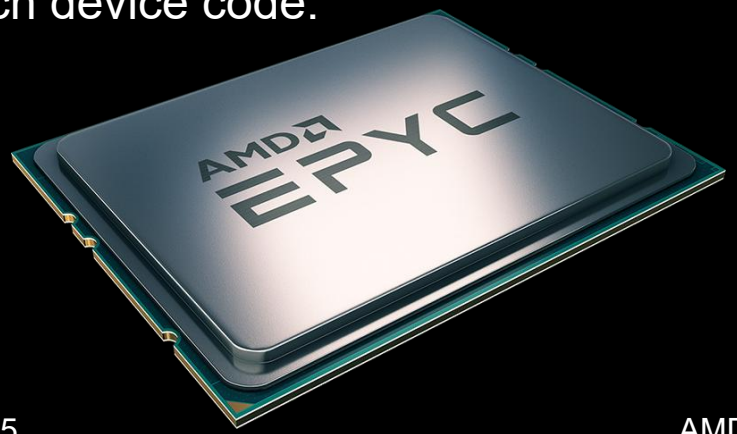
- Extreme Compute Workloads
- Suitable for typical AI work
- Other work entirely on GPU


# The APU programming model

# A tale of Host and Device

Source code for CPU-GPU systems has two flavors: Host code and Device code

- The **Host** is the **CPU**
  - MI300A: 3 “Zen 4” based chiplets
- Host code runs here
- Usual C++ or Fortran syntax and features
- Entry point is the *main* function / PROGRAM
- HIP API or OpenMP® can be used to create device buffers, move between host and device, and launch device code.




- The **Device** is the **GPU**
  - MI300A: 6 XCDs with 38 compute units each (228 CUs total)
- Device code runs here
- C-like syntax (HIP)  or OpenMP directives
- Device codes are launched via *kernels*
- Instructions from the Host are enqueued into *streams*

# Memory model

***Definition***

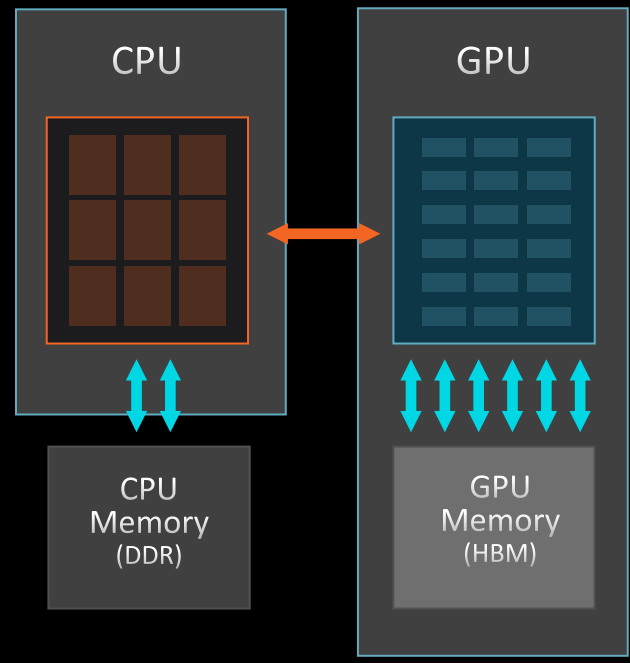
A memory model defines the rules for the synchronization of memory modifications between threads, compute hardware and cache.

A memory model is critical for parallel computing to help both system developers and application programmers avoid data hazard or race conditions where memory is modified by one entity, but another compute unit fails to get the updated value.

# AMD discrete GPUs and memory addressing

In discrete GPU systems, CPU and GPU memory spaces are **separate** and data needs to be **moved** between the two spaces. This data movement can be performed in two ways:

1. By the **programmer**, explicitly
2. By the **Operating System** (OS), who helps move pages on access and subsequent page fault
   - We call this managed memory – short for "memory is managed by the operating system"
   - If no corresponding address is found, the program will fail with a segmentation fault




AMD MI200 GPUs (MI210, MI250, MI250X) and MI300X are **discrete** GPUs

- Implement **managed memory**
- To enable managed memory, `export HSA_XNACK=1`

# XNACK -- what is it and how to use it

## Definition

**XNACK** refers to the AMD GPU's ability to retry memory accesses that fail due to a page fault.

### xnack environment variable

On MI200 and MI300 series GPUs, it can be enabled on a per-process basis using the environment variable HSA_XNACK=1 and disabled using HSA_XNACK=0. Default decided at boot time.

### xnack compiler flag

Run `rocminfo | grep xnack` to check if xnack is enabled

Compilation mode that can assume three possible values: xnack+, xnack-, xnack any.

To change the xnack compilation mode of a program, xnack+ or xnack- may be appended to the architecture flags:

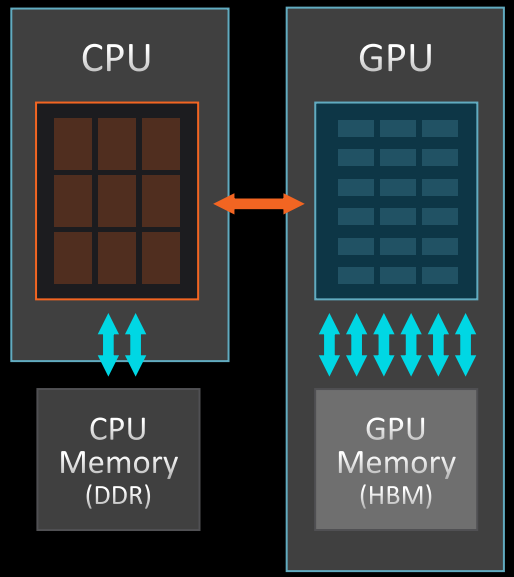
```
--offload-arch=gfx90a:xnack+
```

Supplying multiple xnack options will yield a "fat-binary" with both modes enabled.

When not specified, the default **xnack any** mode will be used.

Code compiled with **xnack any** will run in any case.

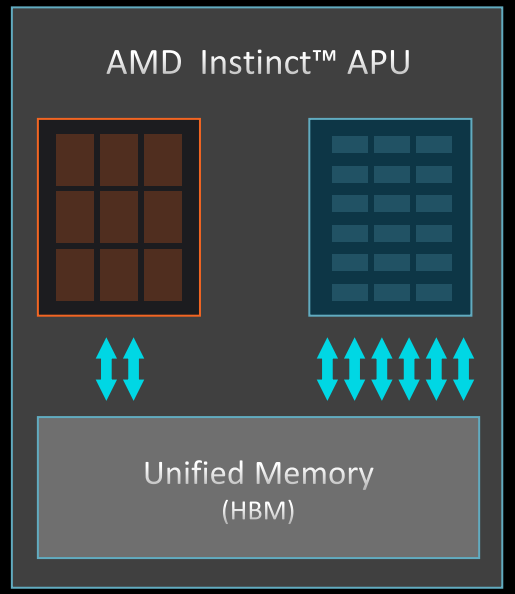
# APU ARCHITECTURE BENEFITS FOR CPU TO GPU PORTING




AMD DISCRETE GPUS

## AMD CDNA™ 3 Unified Memory APU Architecture

- **Eliminate Redundant Memory Copies**
- **No programming distinction between CPU and GPU memory spaces**
- **High performance, fine-grained sharing between CPU and GPU processing elements**
- **Single process can address all memory, compute elements on a socket**
- **Allows incremental porting**




AMD MI300A

Fortran example, analogous for C++

# APU PROGRAMMING MODEL WITH OPENMP®

## CPU CODE

```
!allocation on host
ALLOCATE(var(1:N))

!compute on host
!$omp parallel do &
!$omp  private(i), shared(var)
DO i=1,N
   var(i) = …
END DO
!$omp end parallel do
!sync barrier at omp end …
…
!deallocation
DEALLOCATE(var)
```

## GPU CODE

```
!allocation on host
ALLOCATE(var(1:N))

!compute on device, expl. mem movement!
!$omp target teams distribute parallel do &
!$omp  map(tofrom:var) private(i),shared(var)
 DO i=1,N
   var(i) = …
 END DO
!$omp end target teams distribute parallel do
!host-device sync barrier at omp end …
…
!deallocation
DEALLOCATE(var)
```

Costly to switch CPU -> GPU!

## APU CODE

```
!$omp requires unified_shared_memory
!allocation of unified memory
ALLOCATE(var(1:N))

!compute on device, no expl. mem movement!
!$omp target teams distribute parallel do &
!$omp             private(i),shared(var)
 DO i=1,N
   var(i) = …
 END DO
!$omp end target teams distribute parallel do
!host-device sync barrier at omp end …

!deallocation of unified memory
DEALLOCATE(var)
```

Cheap CPU -> GPU with APU

- Compute kernel
- Special directive to enable unified memory
- Explicit memory management between CPU & GPU -> not needed for APU!
- Synchronization Barrier

# APU PROGRAMMING MODEL WITH HIP

C++ example, Fortran only possible with C bindings and Interface to C for GPU kernels

## CPU CODE

```
double* in_h = (double*)malloc(Msize);
double* out_h =
(double*)malloc(Msize);



for (int i=0; i<M; i++) // initialize
   in_h[i] = …;

cpu_func(in_h, out_h, M);



for (int i=0; i<M; i++) // CPU-process
  … = out_h[i];
```

## GPU CODE

```
double* in_h = (double*)malloc(Msize);
double* out_h = (double*)malloc(Msize);
hipMalloc(&in_d, Msize);
hipMalloc(&out_d, Msize);


for (int i=0; i<M; i++) // initialize
   in_h[i] = …;
hipMemcpy(in_d,in_h,Msize,hipMemcpyHostToDevice);
gpu_func<<< >>>(in_d, out_d, M);
Synchronization implied, hipMemcpy blocks
hipMemcpy(out_h,out_d,Msize,hipMemcpyDeviceToHost);

for (int i=0; i<M; i++) // CPU-process
  … = out_h[i];
```

## APU CODE

```
double* in_h = (double*)malloc(Msize);
double* out_h = (double*)malloc(Msize);



for (int i=0; i<M; i++) // initialize
   in_h[i] = …;

gpu_func<<< >>>(in_h, out_h, M);
hipDeviceSynchronize();

for (int i=0; i<M; i++) // CPU-process
  … = out_h[i];
```

- Compute kernel
- GPU memory allocation on Device -> no copies for host and device on APU!
- Explicit memory management between CPU & GPU -> not needed for APU!
- Synchronization Barrier

# PROGRAMMING ACROSS FRAMEWORKS/COMPILERS

## OpenMP® CODE

```
#pragma omp requires unified_shared_memory

double* in_h, *out_h;
in_h  = new (std::align_val_t(128)) double[N];
out_h = new (std::align_val_t(128)) double[N];


for (int i=0; i<N; i++) // initialize
   in_h[i] = …;


#pragma omp target
{ … }


for (int i=0; i<N; i++) // CPU-process
  … = out_h[i];

delete[] in_h; delete[] out_h;
```

## RAJA CODE

```
double* in_h, *out_h;
in_h  = new (std::align_val_t(128)) double[N];
out_h = new (std::align_val_t(128)) double[N];


for (int i=0; i<N; i++) // initialize
   in_h[i] = …;


RAJA::forall< exec_policy >(arange, [=]
(int i) { … } );


for (int i=0; i<N; i++) // CPU-process
  … = out_h[i];

delete[] in_h; delete[] out_h;
```

## KOKKOS CODE

```
double* in_h, *out_h;
in_h  = new (std::align_val_t(128)) double[N];
out_h = new (std::align_val_t(128)) double[N];


for (int i=0; i<N; i++) // initialize
   in_h[i] = …;


Kokkos::parallel_for(N, [=] (const  int
i){ … };
Kokkos::fence();


for (int i=0; i<N; i++) // CPU-process
  … = out_h[i];

delete[] in_h; delete[] out_h;
```

- Compute kernel
- ~~Device specific memory allocator~~
- ~~Explicit memory management between CPU & GPU~~
- Synchronization Barrier


# Choices in programming languages

# Multiple language paths for AMD GPUs -- all offer portability

Native or Low-level languages — ROCm™, HIP, OpenCL

Pragma-based languages — OpenMP, OpenACC (HPE)

Higher Level Performance Portability languages – Frameworks — Kokkos, RAJA

Standard Based Languages — C++ (HIP) Standard Parallelism

# Pragma-based languages

## OpenMP® – primary supported option for AMD GPUs

- Implemented through LLVM™
- Implemented in the cray compilers
- ➢ **Recommendation**: Best support for portability, easy to get started with

## OpenACC – Supported through Cray, LLVM™ (CLACC) and GCC compilers

- Not as well supported as other options
- ➢ **Recommendation**: Try if your code already has OpenACC implemented
- ➢ Also available: source-to-source translation tools from OpenACC to OpenMP (Intel® and CLACC)

## do concurrent (Fortran only)

- requires rewriting the “do”
- Not supported by all compilers yet, but expected very soon
- ➢ **Recommendation**: Good option to offload very simple loops, can be considered

# Native or low-level languages

Heterogeneous Interface for Portability (HIP)

- A portable layer on top of ROCm and CUDA

Requires a different source on CPU and GPU

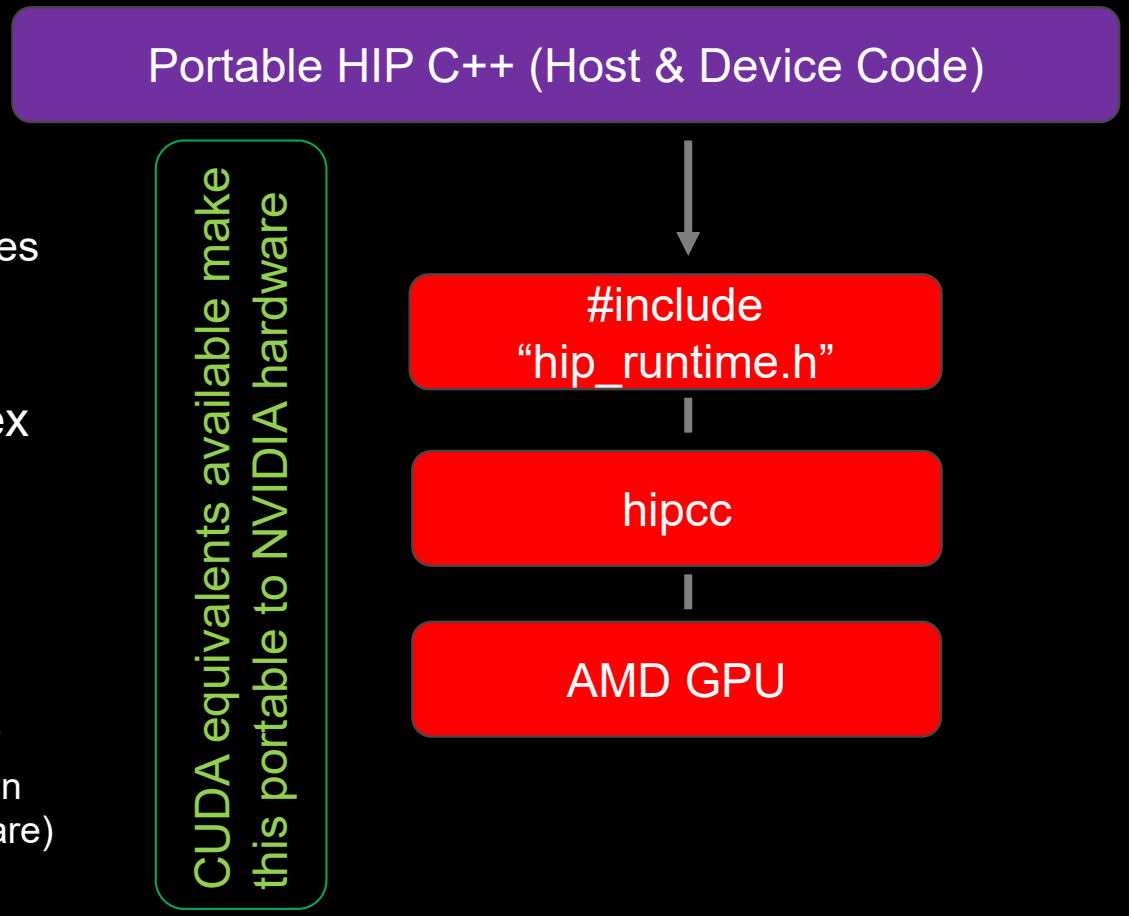
- Larger effort for porting and overhead
- No equivalent of CUDA Fortran available: Fortran requires C interfaces, thus even larger overhead and two programming languages for same app

➢ **Reccomendation**: HIP for *hottest loops* and complex kernels, if not already CUDA ported

➢ If already CUDA ported: Converting CUDA to HIP is straightforward

- Hipify scripts do majority of the work
- Still requires optimization effort to get best performance
  - e.g. a wavefront has 64 threads executing the same instruction (different compared to 32 threads per warp on NVIDIA hardware)

- There are other low-level languages such as OpenCL™



# Higher level performance portability frameworks

## Kokkos – Sandia National Lab (SNL) C++ performance portable programming model

- The Kokkos team has aggressively developed support for AMD GPUs via a HIP backend
- Kokkos handles many of the unique attributes of the AMD GPUs for you
- Parts being integrated into the C++ standard

## RAJA – Lawrence Livermore National Lab (LLNL) C++ performance portability layer

- Modular in structure with separation of compute and data management
- Supports AMD GPUs
- Key kernel patterns have been optimized by AMD

## Advantages of Performance Portability Frameworks

- True single-source application code (at least if you restrict yourself to C++)
- Many of these framework support both CPUs and GPUs

# C++ standard based languages

With the C++ 17 standard, support for parallelism was introduced. The application developer specifies parallelism as the first parameter to a C++ algorithm

- `std::execution::seq` – Sequential execution
  - All operations on the thread that invoked the algorithm
- `std::execution::unseq` – Vectorized execution (C++20)
  - Indicate that a parallel algorithm's execution may be vectorized, e.g., executed on a single thread using instructions that operate on multiple data items
- `std::execution::par` – Parallel multithreaded execution
  - Parallel execution allowed. Operations are indeterminately sequenced within a thread
- `std::execution::par_unseq` – Parallel multithreaded and vectorized execution
  - The various operations can be interleaved with each other on the same thread. Any given operation may start on a thread and end on a different thread

➢ With the release of ROCm 6.1, C++ standard parallelism is available for AMD GPUs.

- To enable, use the `--hipstdpar` compile flag
- The ROCm 6.1 release only supports the `par_unseq` execution policy
- **Recommendation**: Consider this option only for experimental use and with experienced C++ developers, not best performance at the moment

# Other languages

Many other languages also work on AMD GPUs to some level and are continually improving

- SYCL
- Julia: instructions on how to add Julia as a module.
- Python™

ML/AI -- Support for these languages is excellent and portable

- PyTorch – AMD is a founding member of the PyTorch Foundation
- JAX
- Cupy
- TensorFlow
- And many other ML/AI packages
  - See https://github.com/ROCm

# Hands-on exercises

Located in our HPC Training Examples repo:

https://github.com/amd/HPCTrainingExamples

A table of contents for the READMEs if available at the top-level **README** in the repo

Relevant exercises for this presentation located in **ManagedMemory** directory.

Link to instructions on how to run the tests: **ManagedMemory/README.md**

Log into the AAC node and clone the repo:

```
ssh <username>@aac6.amd.com -p <port number>
git clone https://github.com/amd/HPCTrainingExamples.git
```

# Disclaimer

The information presented in this document is for informational purposes only and may contain technical inaccuracies, omissions, and typographical errors. The information contained herein is subject to change and may be rendered inaccurate for many reasons, including but not limited to product and roadmap changes, component and motherboard version changes, new model and/or product releases, product differences between differing manufacturers, software changes, BIOS flashes, firmware upgrades, or the like. Any computer system has risks of security vulnerabilities that cannot be completely prevented or mitigated. AMD assumes no obligation to update or otherwise correct or revise this information. However, AMD reserves the right to revise this information and to make changes from time to time to the content hereof without obligation of AMD to notify any person of such revisions or changes.

THIS INFORMATION IS PROVIDED ‘AS IS.” AMD MAKES NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE CONTENTS HEREOF AND ASSUMES NO RESPONSIBILITY FOR ANY INACCURACIES, ERRORS, OR OMISSIONS THAT MAY APPEAR IN THIS INFORMATION. AMD SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, OR FITNESS FOR ANY PARTICULAR PURPOSE. IN NO EVENT WILL AMD BE LIABLE TO ANY PERSON FOR ANY RELIANCE, DIRECT, INDIRECT, SPECIAL, OR OTHER CONSEQUENTIAL DAMAGES ARISING FROM THE USE OF ANY INFORMATION CONTAINED HEREIN, EVEN IF AMD IS EXPRESSLY ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Third-party content is licensed to you directly by the third party that owns the content and is not licensed to you by AMD. ALL LINKED THIRD-PARTY CONTENT IS PROVIDED “AS IS” WITHOUT A WARRANTY OF ANY KIND. USE OF SUCH THIRD-PARTY CONTENT IS DONE AT YOUR SOLE DISCRETION AND UNDER NO CIRCUMSTANCES WILL AMD BE LIABLE TO YOU FOR ANY THIRD-PARTY CONTENT. YOU ASSUME ALL RISK AND ARE SOLELY RESPONSIBLE FOR ANY DAMAGES THAT MAY ARISE FROM YOUR USE OF THIRD-PARTY CONTENT.


AMD